$2.50

Popular Computing

57

December 1977 Volume 5 Number 12

Coding Fun: Rearranging All The Numbers and profit

Contest 15

Coding Fun: and profit! Rearranging All The Numbers

For our 15th contest, we present another Rearranging problem. See the diagram on the cover.

Start with all the natural numbers, beginning with 3. At all stages, call the number at the head of the stream the leader (thus, at the start, the leader is 3). The procedure to be followed has two alternating phases:

Phase 1. Reverse all sets of K numbers, where K is the value of the leader. Thus, in going from the first to the second line in the diagram, all sets of 3 numbers have been reversed, as shown by the vertical dividers.

Phase 2: Knock out the number that is K numbers away from the leader, and replace it with the leader. Thus, between the second and the third lines on the diagram, the leader (5) knocks out the 6 and replaces it.

All sets of 4 are then reversed, and the new leader (7) knocks out the 5 and replaces it.

The numbers that are knocked out (the circled numbers in the diagram) are permanently extracted from the stream. It is those numbers we want; that is, a list that begins 6, 5, 26, 14, 16, 9, 11, 30, 4, 92, 31, 64, 28, 44, 46,... and the winner of the contest will be the person who (A) provides a long list of those numbers (but not necessarily the longest list) and (B) provides the best method, code, and discussion of the solution, in the opinion of the judges.

There will be just one prize: a TI-58 programmable calculator. Entries to Contest 15 must be received by March 15, 1978. Address POPULAR COMPUTING, Box 272, Calabasas, California 91302. □

PROBLEM 212

POPULAR COMPUTING is published monthly at Box 272, Calabasas, California 91302. Subscription rate in the United States is $20.50 per year, or $17.50 if remittance accompanies the order. For Canada and Mexico, add $1.50 per year to the above rates. For all other countries, add $3.50 per year to the above rates. Back issues $2.50 each. Copyright 1977 by POPULAR COMPUTING.

Editor: Audrey Gruenberger
Publisher: Fred Gruenberger
Associate Editors: David Babcock, Irwin Greenwald
Contributing editors: Richard Andree, William C. McGee, Thomas R. Parkin
Advertising Manager: Ken W. S[illegible]
Art Director: John G. Scott
Business Manager: Ben Moore

5 How to produce garbage

5.1 WHY ARE WE DISCUSSING GARBAGE?

The term "garbage" in computing means unwanted information, garbled information, or incorrect information in any sense. Computer people are fond of citing the GIGO principle—"Garbage In, Garbage Out." That is simply a terse way of saying that output is always a function of input, and implies that the results flowing from a computer can never be better than the data moving into it.

All of which is true, but it does not go far enough. The *program* that processes the data is also a form of input. Thus, even with the best possible data, it is still not difficult to produce garbage.

If there is good data and a program that is logically correct, thoroughly debugged and tested, the end result can still be garbage. (This does not even count machine failure, which is rare, but not impossible.)

What does all this have to do with the social effects of computers? Just this: People have a tendency to wave their arms and cry, "Why don't they use computers to do this or that?" without considering all the implications of their plea. As a practicing computer man, I am all for extending the use of the computer to new applications in the service of man, since the goal of almost every human endeavor is to help make people happy. Let us consider some of the many ways that garbage can be, and has been, produced on a computer.

Chapter 5 is from the book

Computers and the Social Environment

Published by Melville Publishing Company.

Reprinted by permission of John Wiley & Sons, Inc.

5.2 FAULTY LOGIC

We used the phrase "a program that is logically correct and thoroughly debugged and tested" a few paragraphs back. It would be nice to have such programs. Unfortunately, except for very small programs, programs for trivial problems, or programs whose results can be independently tested, there are few that can ever be said to be thoroughly and completely tested. Every decision point in a program (the instructions that implement the flowchart diamond) can multiply the number of possible paths through the program by two. Thus, a program that has 20 decision points—a relatively small program—can have a million possible paths through it, and it is just not feasible to test every one of those paths. We have a lot of faith that if we test the logical action of each decision point independently, and a few dozen typical paths in concert, that the entire program will probably function accurately. But consider how weak this faith is when the number of decision points gets to 500 (a modest number for a payroll program, for example). The number of possible paths can now be 2^{500}, which is 3.27339×10^{150}. Clearly, not every path of such a program can be explored, even by the computer itself.[1]

Experience has shown that most logical troubles in computer programs occur at the decision points, but there can be other troubles, even such simple ones as writing ADD when one means SUBTRACT. The task of locating all such logical errors may be beyond human capability, although we certainly try our best, and many programs do seem to function properly eventually. But most programs of any size (say, 2000 instructions or more) can contain subtle errors. The errors can lie in wait for years, and cause great embarrassment when finally revealed.

There are well-established techniques for testing large programs. For example, the program can be written in small logical modules, independent of each other, and these modules can be tested separately; by definition, each module is a short program, and therefore, tends to be easier to test thoroughly. Even so, the resulting large program, made up of the sum of the various modules, must be tested as a whole. The biggest factor that operates to reduce the likelihood that the testing process will be carried out carefully is the pressure exerted on the programmer to produce useful results. At the time when the programmer is called on to do his best work, the promised program is often seriously late, and there is a demand for some output. The catch phrase in the trade is "Why is there always time to do it over, but never time to do it right?"

Thus, our first big source of garbage is the computer programs themselves. Programming is an imperfect art; we have a long way to go before we can produce programs in which we have a high degree of confidence.

[1] If we could explore one path completely in one microsecond, the total exploration might take over 10^{137} years.

5.3 PACKAGED ROUTINES

Few, if any, programmers ever write all the instructions for their own programs; they invoke packaged routines to perform common functions. At the lowest level, every program calls for input in decimal or alphabetic form, and produces output in the same form. But, since the machines function in binary, each vendor furnishes input/output routines to make the necessary conversions. *Those* programs are subjected to stringent tests thousands of times a day. There are hundreds of such packaged subroutines available to a programmer, including:

1. Function routines, to calculate roots, logarithms, exponentials, trigonometric functions, and inverse trigonometric functions.
2. Extended arithmetic routines, for multiple precision work, matrix arithmetic, complex arithmetic, etc. (If these terms are not familiar, do not worry about it.)
3. Random number generating routines.
4. Utility routines for sorting, merging, searching, and report writing functions.

Every one of these packaged routines is subject to the same kind of errors described in the preceeding section. Programmers are particularly gullible about assuming that any routine that is in their library must be perfect; it is almost unheard of for a programmer to question the validity of a packaged routine, much less to take the trouble to test it for himself. Thus, our troubles may become compounded.

But let us assume that the packaged routines do exactly what their authors claim they should do. That is not at all the same as saying that the routines will do what the user expects them to do. Let us see if we can make that idea clear.

Suppose, for example, we make use of a library subroutine for SINE(X) in a problem we are working on. The assumption is that if we feed the routine a number, X, we will get back the sine of X, which should be a number in the range between plus and minus one. That is exactly what the author of the SINE routine intended. We

proceed to feed the routine a value of X, 37852, and the "sine" comes back as 38.27406. We are not supposed to use values of X greater than 6.283185; the routine was written to operate properly only up to 2π (radians). Does it say so? Probably it does, in some obscure piece of documentation that is long since buried.

The example just cited was all too common around 1965. It is rarer today, and almost unheard of for elementary functions like SINE.

Perhaps that example is too mathematical for you. Consider, then, a packaged random number generator, which is the essence of any work in sampling or simulation. We are told that such a package is in our library, and we need only write RND in our program, and the package will furnish us, on each execution of that instruction, a fresh random number. Would you trust that? Suppose the generator produces 4096 numbers and then starts the same sequence all over; will that meet the needs of your program? Suppose every number produced by the generator is divisible by 17; might that disturb your results?

Here is one more example. We might invoke a packaged routine to merge two streams of input data. The merge routine rests on the assumption that both input streams are in sequence, since that notion is implicit to the process of merging. Your input data has a sequence error, and the packaged merge routine merrily proceeds to produce a mess for you. Who is to blame here?

5.4 SYSTEM ERRORS

Our programs must be fed through an operating system to become operational and must interface somewhere with people. The operating system called OS/360 is known to have hundreds of bugs, every one of which may affect the running of your program. A discussion of this particular level of trouble would be highly technical. Let us just leave it at this: operating systems are not noted for helping us as much as they are notorious for causing frustration and agony. Nevertheless, we are stuck with them, and we learn to live with their curious idiosyncrasies and to program around them. They seem to be a necessary evil, but few kind words are recorded about them.

If you or a friend has access to a large computing installation, try the following experiment: Select a higher level programming language (Fortran, COBOL, PL/I, etc.) and prepare a program of exactly one instruction: namely, HALT, in whatever terms that language requires. This program (which will be punched on just one card, and which should ultimately produce a machine-language program of exactly one word) will have to be surrounded by the required Job Control Language to get it through the operating system. Run the program, and examine carefully what you get back from the machine in printed output. Better yet, have an expert try to explain to you the meaning of the various pages of printout you receive (up to as many as 15 *pages* of information) for your one-instruction program. Note in particular any printed information that your expert cannot account for and cannot explain to you. If you then ask him why *you* are being given information that *he* cannot understand, the following will probably ensue:

"Don't worry about that. . . . We always get that information, but we don't know why. . . . We could probably get rid of it, but then we would probably have some other things work badly, so we'll live with it. . . . Why don't *you* live with it? Just ignore it, and don't be a chronic complainer."

A more sophisticated experiment is the following: Write a program (which can be intrinsically meaningless) in some higher level language that deliberately violates the rules of that language, and see what the system does with it. A famous program along these lines is the following in Fortran:

```
      DØ 27 J=1, 10
      J=J+1
      PRINT 8, J
 8    FØRMAT (I9)
27    CØNTINUE
```

which violates the rule that says that one should not alter the index of a DØ-loop within the loop. But the index is altered in the second statement. What would you expect Fortran to do? What would you want Fortran to do with that program? If you have access to a Fortran compiler, try it.

5.5 DATA

This brings us to the final source of garbage: the data of the problem. If the output is no better than the input, then faulty data *guarantees* faulty results.

How is input data made into machine-readable form? The commonest method is by keystroking, wherein handwritten, typed, or printed information is transcribed manually. If the source data is illegible or the transcriber is tired, then the likelihood of errors at this stage increases.

We can shortcut that process somewhat by having the data recorded

directly in machine readable form, by using mark-sensed cards, a Porta-punch, or an optical scanner. The encoded numbers on bank checks and deposit slips are scanned magnetically. Even if keystroking is used, we can protect ourselves by having the data punched several times by different people, checking the resulting card decks for agreement. In some systems, check digits are added to part numbers and account numbers to insure better keystroking.

However, none of these measures protect us from data that is simply wrong. If a student's test grade is 85, but I write 65 in my grade book, then I have a data error that will not be revealed by any method. If I record a grade of 85 as 185, we might find the error, provided that the program does some extensive editing on the data. But that, too, is rare. Programmers are quite glib about making their programs edit the input data, until it comes to actually doing it, and then they exhibit more of that faith which says that there is a special providence for fools, drunks, and careless programmers.

The absence of reasonable tests in a program leads to foolish results. If a program calls for an input number that should be the temperature of the river in degrees Fahrenheit, then a value of 378 is simply not possible, despite the neatness with which holes are punched in a card. A program that sensibly edits the input data will promptly reject such a value. But it can go the other way. Suppose an input temperature is assumed by the programmer to be in the range from 40 to 99 degrees (and his program thus reads only two decimal digits on an input card), but a case comes along for which the temperature is correctly 163 degrees—his program reads this as 63. Once again, the program will charge merrily along with incorrect data.

Data errors can be quite subtle. The originator of a problem usually has clearly in mind the nature of the data involved, its range, and its precision. He may forget to mention these vital facts to the person who writes a program for him, and the programmer may forget to ask or, worse, may make his own assumptions about the nature of the data.

5.6 SUMMARY

This chapter may have given you a discouraging picture. The public generally regards computers as being infallible, and the people who run them as not only infallible, but omniscient and fantastically clever. The machines are quite good but not infallible; the people are, perhaps, in the upper 10 percent of the intelligence scale. But they are human, and emotional, and irrational, and they make mistakes. The computer is a machine that is delivered from the factory as a useless device. It is turned into a useful device by a program, and therein lies the trouble. There are endless opportunities for error in the writing of even a modest program. In computing, McDougall's Law operates with devastating effects: If anything can go wrong, it will; and it will happen to you, and at the worse possible time.

Therefore, when we proceed to examine how computers are being used to solve problems in daily life, keep in mind the following points:

1. While computers are not likely to make mistakes, people are. And people control the computers.

2. Any proposed computer application will cost more than its original estimates, it will take longer to get running than was first planned, and it will not work as well as its proposer had in mind. Computer people are notorious for guessing wrong on all three of these items by very large factors.

3. Every nontrivial program probably contains at least one solid bug, which may go undetected for years.

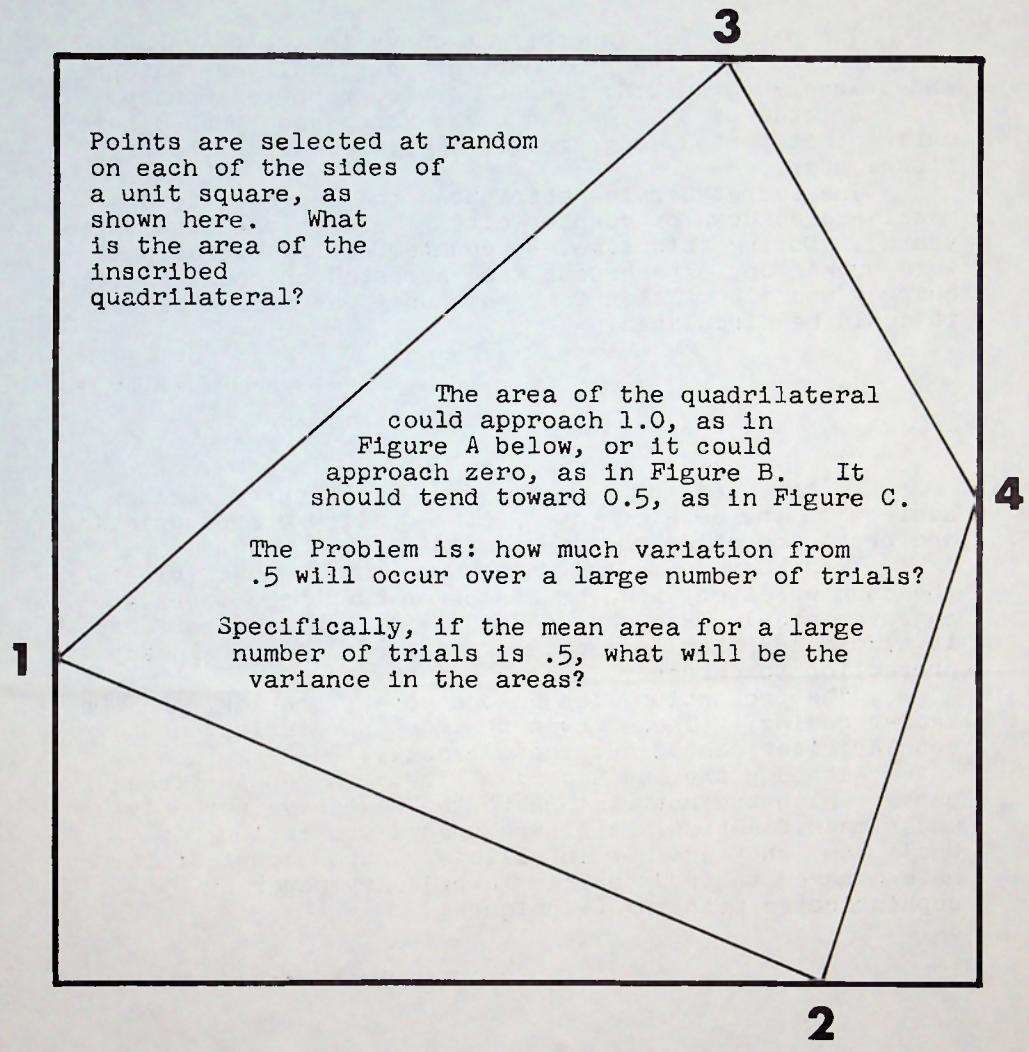
4. Any fool can think of wonderful ways to use a computer, but it takes intelligence to think of the consequences, the overall costs, and the possibilities for eventual trouble.

Exploring Random Behavior -- 4

So you think you understand how random processes will behave?

PC57--7

PROBLEM 213

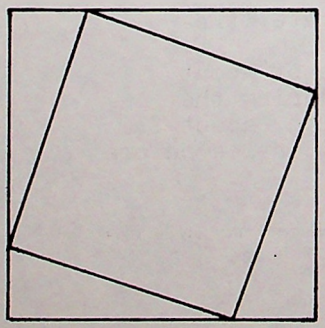


Points are selected at random on each of the sides of a unit square, as shown here. What is the area of the inscribed quadrilateral?

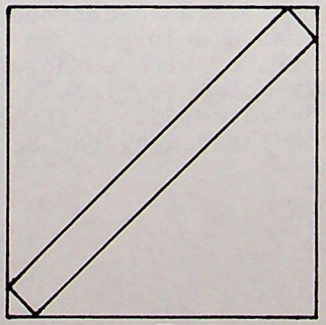
The area of the quadrilateral could approach 1.0, as in Figure A below, or it could approach zero, as in Figure B. It should tend toward 0.5, as in Figure C.

The Problem is: how much variation from .5 will occur over a large number of trials?

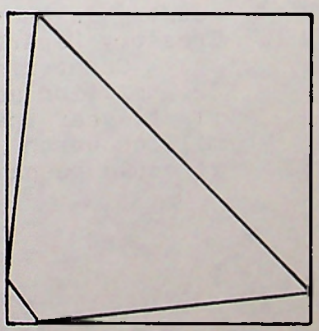
Specifically, if the mean area for a large number of trials is .5, what will be the variance in the areas?



A



B



C

PC57--8

COUNTERFEITING

The passing of counterfeit money in the Los Angeles area dropped more than 30% during the fiscal year which ended June 30, the U.S. Secret Service reported Monday.

A total of $301,000 in bogus money was passed here during that period, compared to $440,000 during the prior fiscal year.

The Secret Service attributed the decrease to a massive crackdown on counterfeiting here in the last three years. During this time, 39 counterfeiting operations were broken up, 575 persons were arrested on counterfeiting charges and $16 million in bogus money was seized before it could be circulated.

The Los Angeles *Times*, August 15, 1973

...the need for such a system (Money Scan) became acute with the development of high quality offset printing and precision color photocopy machines.

These developments, which are readily available to the counterfeiter, mean he no longer must make intricate engraved steel plates by hand to print good counterfeits. It also means that intricacy of design no longer is any protection to currency.

"The protection lies in the paper, the ink, and the secret coding. These, even the most sophisticated counterfeiter cannot duplicate exactly."

Although the red and blue fibers in genuine Treasury paper, which are not visible to the naked eye but only under magnification, still are a good protection for Uncle Sam, they are not infallible. Some counterfeiters have managed to produce them in ordinary paper by sophisticated printing techniques.

The Los Angeles *Times* (UPI) June 12, 1972

The amount of fake money circulating in this country is increasing, but at a decreasing rate, the Treasury Department has disclosed.

Counterfeit currency with a face value of about $2.9 million actually got into circulation during the fiscal year that ended June 30. In addition, about $12 million worth was confiscated by the Secret Service before it could be passed.

The Washington *Post*, July 26, 1969

According to Robert Powis, head of the Secret Service here, the courts convict 97.1% of all currency counterfeiters brought to trial.

In the fiscal year ending last June 30, Powis points out, $2.9 million in bogus currency was passed in the United States and an additional $12 million seized before it could be circulated.

The truth is that most bad bills will not survive a second look. Details are blurry, colors flat. Then there is the different feel between genuine and bogus notes.

The amount of bogus money passed in the first six months of the current fiscal year was $1.1 million, down 20% from the same time period last year.

The Los Angeles *Times*, June 7, 1970

Secret Service agents today announced the break-up of a giant counterfeiting ring which printed and passed more than a million dollars in phony $100 bills.

Twenty-five men and women have been arrested, but the counterfeit presses and plates were not located.

Those arrested were charged with printing and uttering more than $500,000 in artfully printed fake money. It was reported that the Chicago-based counterfeiters burned another million dollars' worth of money before agents closed in.

The San Francisco *Chronicle* (UPI) March 6, 1959

The counterfeiter does not have access to equipment as sophisticated as the Government's; therefore, his notes are inferior. Most counterfeits are made by a photo-mechanical process. The printing appears flat and lacks the three-dimensional quality of genuine notes.

Counterfeiting, bulletin of the Secert Service

Since early this year nearly 200 counterfeit 1000-yen notes have been discovered, largely in the area centering on Tokyo.

No one knows how many of the fake bills may be in circulation, however, for the counterfeiters are so good that all but three or four have gone undetected until they have reached large banks.

The bogus bills bear many different serial numbers, which has added to the problems of detecting them.

The Los Angeles *Times*, September 11, 1962

Police said Tuesday they had uncovered an international swindle, centered in England and West Germany, involving forged American Express travelers' checks worth $1 million. The forgeries were described as "virtually perfect."

The Los Angeles *Times* (AP) June 9, 1971

The need for automated detectors is obvious. During 1972, the Secret Service confiscated some $27.7 million in bogus currency--a record. Luckily, 83% was grabbed before circulation, and only $4.8 million is listed as "losses to the public." There's no way of telling how much gets passed undetected, since counterfeiters don't publish annual reports. Says one source: "I think the Secret Service figures are only the tip of the iceberg."

Popular Science, October, 1973

With the exception of the last two sentences quoted above, what we have is patently all nonsense. The Secret Service believes it; perhaps they have to believe it. They have not been misquoted by the press; they have been spouting indentically the same nonsense for forty years or more.

The fight against counterfeiting seems to rest on these tenets:

1. No one but the government can make currency as good as that made by the government.

2. Properly trained people can distinguish unerringly good bills from bad. This includes every employee of the Secret Service.

3. Mechanical safeguards (paper, ink, engraving, red and blue threads) are additive in their efficacy, and enough of them add up to a product that cannot be duplicated.

4. The quantity of counterfeit currency found is an accurate index of the amount of counterfeit currency manufactured. It is also an index of the amount that has already been passed.

And, yes, there is a tooth fairy. Bankers and Secret Service agents have repeated these tenets for so long that they have come to believe them. If you want to prove the converse, try this experiment. Go to a bank and exchange $1 in change for a dollar bill. Then find the manager (or anyone else who thinks he can distinguish good money from bad) and show him the bill, with the suggestion that it may be a phony. And then watch him squirm. The experts can spot obvious counterfeits, but how about ones that are not obvious?

Our paper currency is made by an engraving process; that is, printing with ink on paper. It is high grade printing, but it is a mechanical process, done by humans. What one group of humans can do, in a mechanical way, another group can duplicate, or even surpass. (Recall the Victor Moore movie, in which he made counterfeits that were better than the government's--they wouldn't burn.)

The Secret Service knows all about the counterfeiters they know about; they obviously know nothing about any counterfeiters who make a product that is undetectable. They like to show off to the press the bogus bills they've found, which are invariably clearly fakes. They have yet to show off a catch of mis-made good bills.

Isn't it mathematically obvious that someone has solved that problem and done it? This point can, in fact, be proved. During World War II, the Germans succeeded in duplicating British currency, to the point where the British government had to withdraw all 5-pound notes from circulation.

On that assumption (that someone, or even many someones, is busy producing well-made bills), what possible protection is there?

The safeguarding of negotiable paper should rest on intellectual grounds, not mechanical ones. Each such piece of paper bears a serial number, assigned consecutively during the printing. Suppose it had <u>two</u> serial numbers with one of them the enciphered version of the other. Postulate a cipher system that is well constructed; the normal serial number is itself the local key to the cipher. Thus, given the normal serial number, the other number is derivable from it by a mathematical process, and the details of that process constitute the cipher.

Suppose that the heart of the process is a standard random number generator. As an overly-simple illustration, consider this generator:

$$x_{n+1} = 37^5 \, x_n \bmod 10^8$$

and the normal serial number of a given bill is 98765432; this is x_n in the recursion above. The generator gives an enciphered equivalent of 68487858. (This process can be made as complex as one pleases; the arithmetic will be carried out by a computer.)

The prospective counterfeiter now has only the following choices:

1. He can select any legal combination of serial numbers and replicate that pair for all the bills he prints.

2. He can avoid the issue by printing any good looking numbers.

3. He can duplicate many legal pairs, one by one.

4. He can try to break the cipher system and thus duplicate exactly what the issuing agency does.

Let us dispose of these in turn. Breaking the cipher system is supposed to be impossible, by definition, but suppose it were possible, and the government's system could be duplicated. To say the least, the setup costs for the counterfeiter have been tremendously increased. The probability has been reduced that a criminal group could assemble the printing and engraving skills together with cryptographic experts and computer programmers, all at once. The threshold costs have gone up, and much of the profit and motivation has been taken out of the game.

The same point applies, but slightly weakened, to the third point, that of duplicating legal pairs of numbers. Such duplication would require facilities that would significantly lower the potential profit for the operation.

Points one and two are taken care of by constant monitoring of the documents as they return to (or pass by) the issuing agency. Each pair of numbers can be tested for validity and illegal pairs or duplicates can be detected. To be sure, this requires a large scale sampling procedure on a continuing basis. At a minimum, there would be the following: any bill that does not check out properly is a provable counterfeit, which is more than we have now.

Such a system is not foolproof. The protection it offers is additive to the best that is currently available. The fact of its existence would operate as a deterrent to counterfeiting and would increase public confidence in the negotiable paper it protects.

Money is one form of negotiable paper; it happens to be the one for which there are special laws against counterfeiting, and these laws are stringent. The same principle applies to other paper, such as traveler's checks, for which the federal laws against counterfeiting do not apply (only laws against fraud and embezzlement) and which are easier to copy in any event. □

GAS MILEAGE

A car's gasoline log shows the following entries:

Odometer reading	Gallons to fill tank
1077	4.4
1296	6.6
1510	6.5
1700	6.8
1850	5.3
2070	7.8
2260	7.1
2450	6.8
2610	5.5
2770	5.0

The figure on the right is obtained by halting each fill when the self-serve pump shuts off.

Using all the given data, one possible result would be total miles (1693) divided by total gas (58.4 gallons) for an average of 28.99 miles per gallon. Over the given range of miles, is the car's mileage performance improving or decreasing?

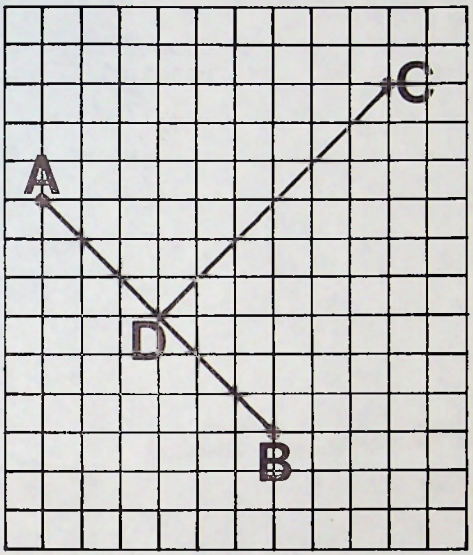
How many miles, on the average, does this car obtain from a gallon of gas?

This should be a trivial problem in arithmetic--just divide the miles traveled by the gallons consumed. One tankful cannot provide a meaningful figure, inasmuch as automatic pumps do not cut off at a uniform point, and the difference from one pump to another could be a gallon or more. Moreover, over the period of one tankful, there is not a sufficient base to estimate the varying conditions of travel (city vs. country, etc.). On the other hand, if one considers too many purchases, then the result is not a measure of what the car is doing now (the engine characteristics may change in 2000 miles). □

NEBULA

Given two points, A and B, on a lattice, a third point, C, can be located so that CD = AB. The point C may be precisely another lattice point. Generally, point C will not turn out to be a lattice point, but when the coordinates of C are calculated they can be rounded in the usual way so that point C can be the nearest lattice point.

For example, with A = (1,9) and B = (7,3) as shown here, the point C becomes (10,12). If, however, B is moved to (9,3), then the coordinates of C become (10.3407, 14.4544), which rounds to (10,14) as the closest lattice point.

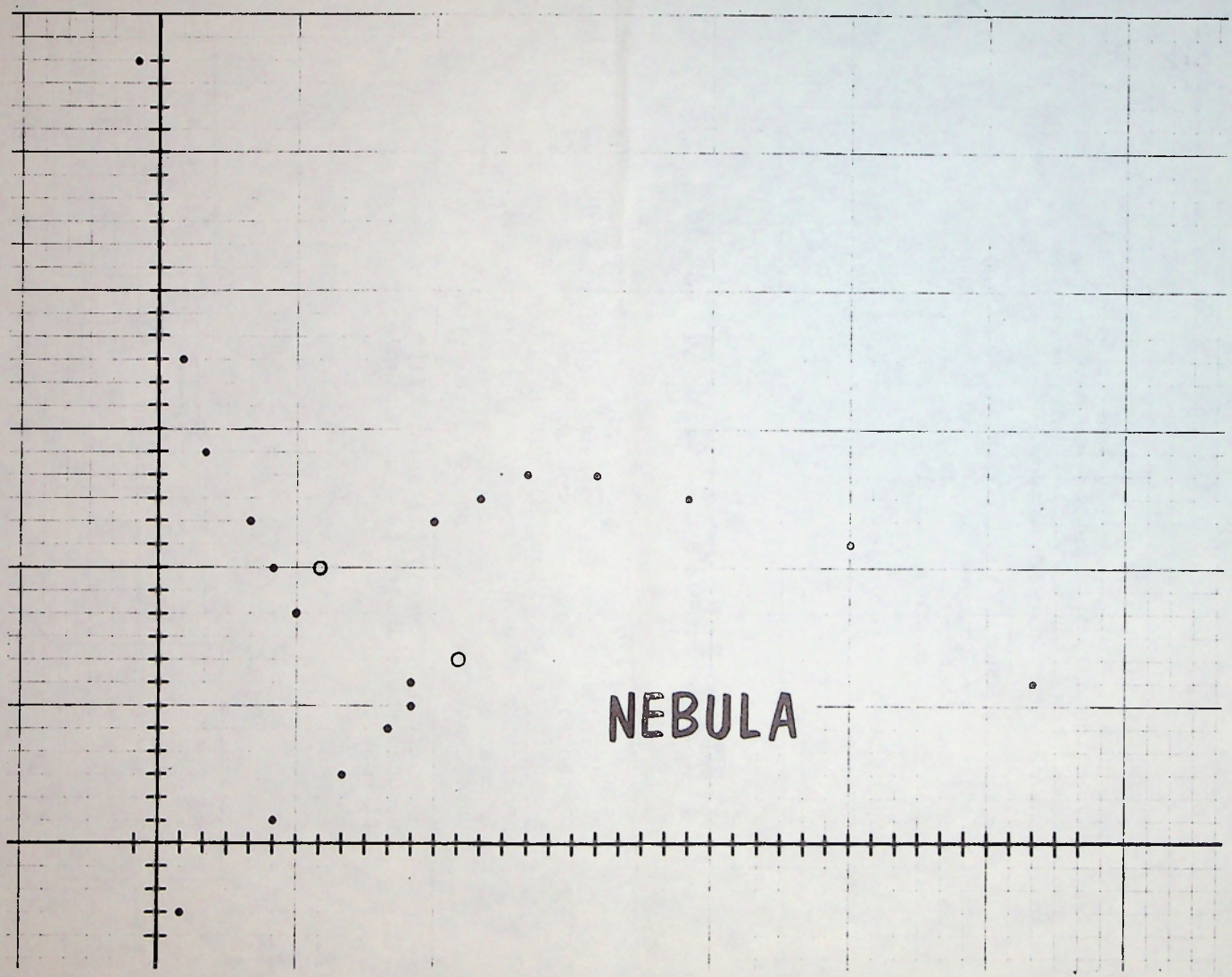


Thus, two given lattice points determine a third. Then, using B and C, a point D can be similarly located, and the process can continue.

Using the points (6,10) and (11,7) as starters, a pattern of new points is generated, as shown on the next page, forming a sort of spiral nebula.

If the distance BC = AB, the spiral would be trivial and dull. If BC is less than AB, the pattern would spiral in. If BC is greater than AB (as when CD = AB), the pattern spirals out, as shown in the accompanying Figure.

Given two points as starters, write a program that will generate the successive points of the nebula (with CD = AB). The problem is interesting in that there are two points at each stage that satisfy the conditions of the problem, but only one of them is acceptable to form the spiral, and selecting the proper points at each stage is the nub of the computing problem.



PC57-15

PATTERN RECOGNITION

PROBLEM 216

In the accompanying Figure there are six 10 x 10 patterns of dots. Two of the patterns are identical. Assuming that the only information you have is the coordinates of the black (or white) dots, how would you isolate the matching pair of patterns?

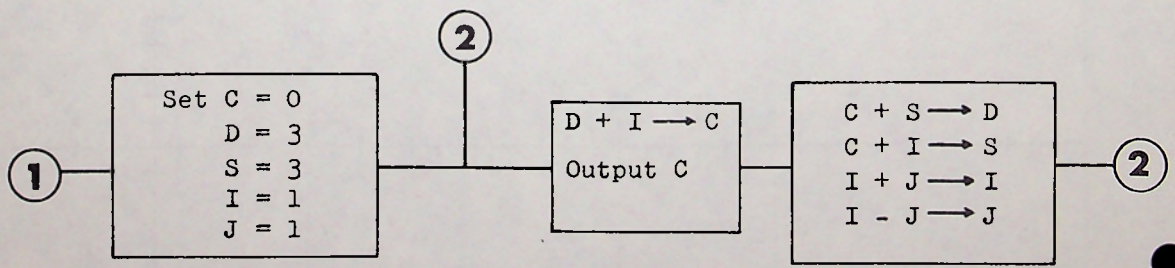
Now suppose you had 10,000 such patterns and you were to find the two that match (assuming that there are two and only two that match)? What would be your plan of attack on such a problem?

(Note: the patterns shown here are rotated but not turned over.)

Problem Solution

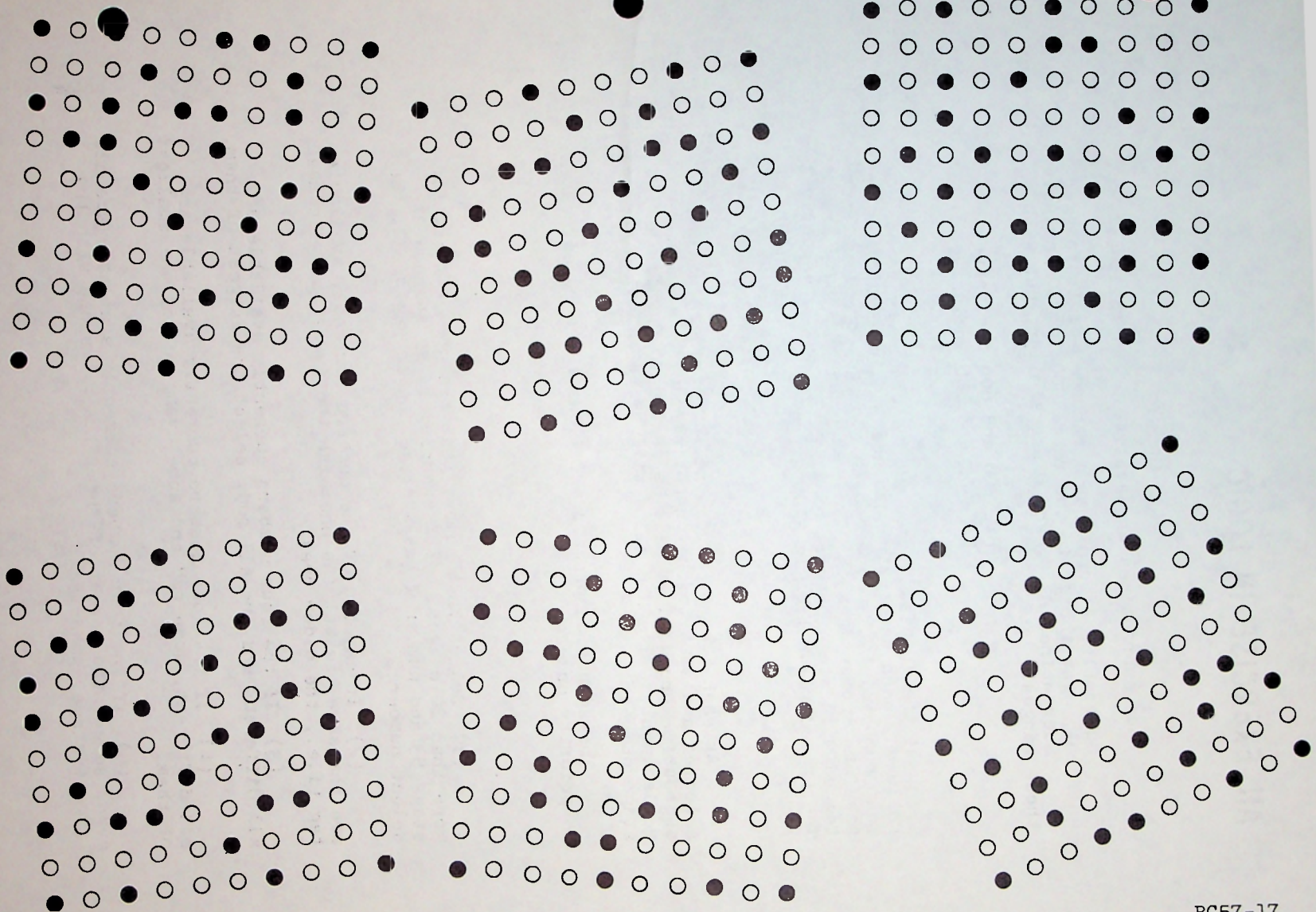
In issue 54, as a preamble to Problem 189, we presented Les Marvin's problem (from the Journal of Recreational Mathematics), which called for the generation of a sequence made up of terms chosen from a set of Fibonacci-like sequences. A flowchart was given for a possible solution.

John D. Beeby, Millbrae, California, has reduced the solution to a much simpler form, in this flowchart:



S, C, and D are successive terms on the same line.

I and J are incrementing variables, working from one line to the next.



PC57-17

PC57-18

AN EXERCISE IN LOGIC

Four subroutines are available, as follows:

Subroutine A furnishes a random 3-digit integer uniformly distributed in the range 000 to 350 inclusive.

Subroutines B, C, and D are similar, but with these ranges:

B: 200 and 600 inclusive
C: 400 and 800 inclusive
D: 700 and 999 inclusive

At the initial stages of the procedure to be followed, each subroutine has been called and there are four numbers, P, Q, R, and S in storage (one from each of the four subroutines). These four numbers have been tagged with stage numbers of 1, 2, 3, and 4 respectively. The procedure will begin with stage 5.

At each stage, one of the four numbers will be selected and sent to the output stream, and whichever subroutine produced it will be called to furnish a replacement. The new value will be tagged with the new stage number.

The selection procedure follows these rules, taken in priority order:

(1) Select that number that has been sitting for more than 20 stages. For example, if we are now at stage 53 and number Q has a stage number tag of 32, then select number Q.

(2) If any two of the numbers currently available are the same, select the one with the lowest stage number tag (i.e., the older one).

(3) If the four numbers currently available are all different and are all odd, select the largest of them.

(4) If the four numbers currently available are all different, some odd and some even, select the smallest of them.

(5) If the four numbers currently available are all even, select the newest of them; i.e., the one with the largest stage number tag.

(6) In all other cases, select the oldest number; i.e., that one of P, Q, R, and S with the lowest stage number.

For example, if the situation is:

Contents ⟶	287	552	480	911
	P	Q	R	S
Stage number ⟶	25	19	17	10

We are at stage 25: P has just been furnished by subroutine A and the selection process is in order. Rule (4) applies, and the number 287 is to be sent to the output stream. Subroutine A is to be called to furnish a new value for P, tagged with stage number 26.

We now have four problems:

(K) Draw a flowchart of the logic of the selection procedure.

(L) Devise a procedure for validating that logic. This procedure should consist of forcing specific and predetermined output for the four subroutines, with the consequent output stream predicted.

(M) Calculate the mean and standard deviation of the expected output stream.

(N) Estimate the frequency of use of each of the four subroutines.

A note on the construction of the subroutines. Subroutine C, for example, is to produce on demand a number at random, uniformly distributed in the range from 400 to 800 inclusive. This means that each of the 401 possible numbers should have an equal chance of appearing at each call of the subroutine, and that the sequence of those numbers should have no discernable pattern.

A fifth subroutine is available, whose output has been certified by some means as a pseudo-random number generator. Suppose that the output of this subroutine is uniformly distributed in the range from zero to 2,147,483,647 (that is, one full word on a 32-bit word machine). Divide a selected random number by 401. The remainder of this division will be a number in the range from zero to 400. Add the remainder to 400; this will produce a number at random in the range from 400 to 800, as required. A similar procedure is used for the other three subroutines.

The entire procedure is artificial and arbitrary, with no practical application. However, it provides an interesting exercise in logic and in the devising of a test procedure for a debugged program. It has been pointed out over and over in textbooks that:

(a) Computers do only what they are instructed to do; no more and no less;

(b) Output is a function of the input data and the instructions;

(c) In theory, anything that computers do could be done by hand.

Such precepts imply rather strongly that computers differ only in degree from other calculating devices. Problems such as the one given here are designed to show that computers differ in kind from other tools; that computers truly offer something new in the world. This problem is serial in nature and also calls for the manipulation of random numbers. The machine lets us play out the consequences of our decisions. Further, it permits us to compound our decisions, to the point where it is fair to say that the effects could not be determined by hand methods. Simple problems can be used to demonstrate that, with a well-defined procedure, it can be difficult, if not impossible, for the author of a program to predict what will happen when the program is executed. □

PROBLEM 218

FUN WITH EQUATIONS

The cubic equation:

$$Ax^3 + x^2 + x - 1 = 0$$

has a real root for any value of A. If A = 1, the root is around .544. Suppose we take that root and use it as the next value of A, and solve again. Then let the root of that equation be the new A, and so on.

This process converges--to what value?

Start over with A = 1. Find the root of

$$Ax^3 + x^2 + x - 1 = 0$$

and <u>double</u> it for the next value of A. This also converges; would you expect it to converge to a larger or a smaller value than in the first case?

Again, starting with A = 1, suppose the root found at each stage is <u>tripled</u> to become the new value of A. Will the convergent value be larger or smaller than in the second case?

We are repeating the same problem with the procedure:

(root)(factor) replaces A.

So far, we have used the values 1, 2, and 3 for the factor. What happens as the factor approaches zero? What happens as the factor becomes very large?